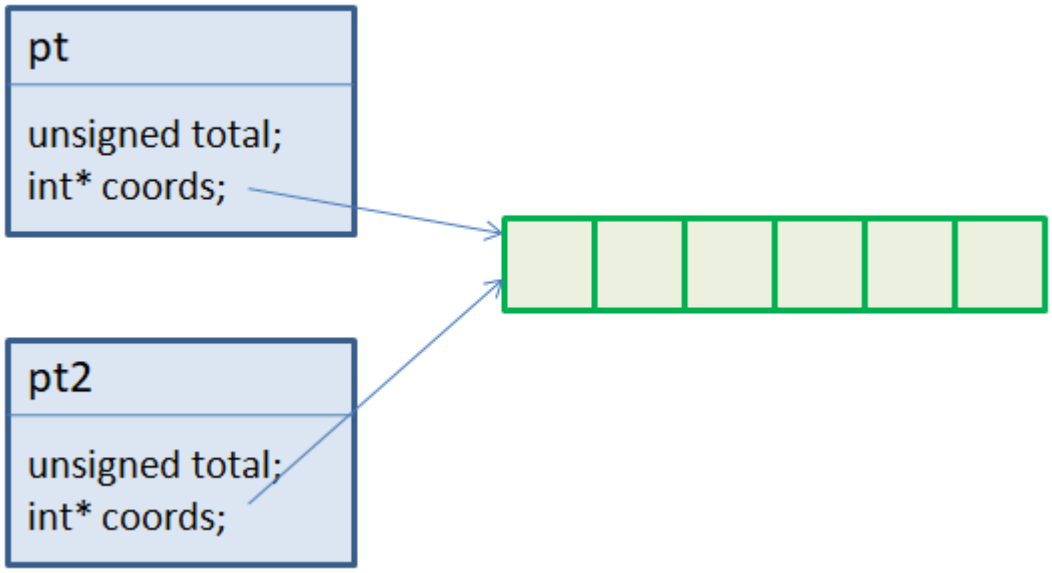
На предыдущем занятии мы с вами получили класс для представления точек в N-мерном пространстве:

```language-cpp
class PointND {
    unsigned total {0};
    int *coords {nullptr};
public:
    PointND() : total(0), coords(nullptr)
        { }
    PointND(unsigned sz) : total(sz)
    {
        coords = new int[total] {0};
    }
    PointND(int* cr, unsigned len) : total(len)
    {
        coords = new int[total];
        set_coords(cr, len);
    }

    unsigned get_total() { return total; }
    const int* get_coords() { return coords; }
    void set_coords(int* cr, unsigned len)
    {
        for(unsigned i = 0; i < total; ++i)
            coords[i] = (i < len) ? cr[i] : 0;
    }

    ~PointND()
    {
        delete[] coords;
    }
};
```

Но остается еще одна проблема его использования. Давайте предположим, что создается новый объект PointND на основе ранее созданного:

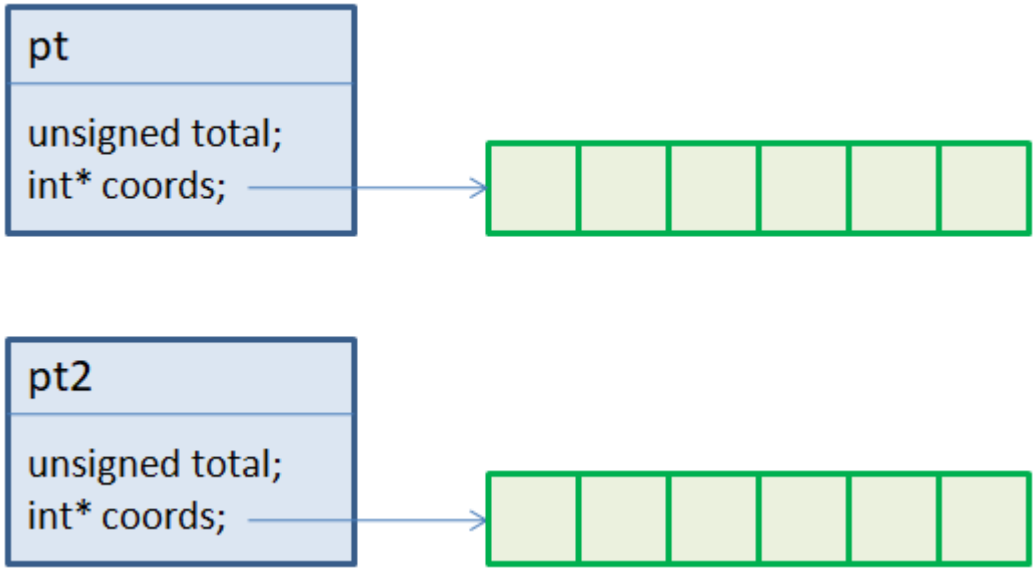
```language-cpp
int main()
{
    PointND pt(5);
    PointND pt2{ pt };

    return 0;
}
```

В этом случае объект pt2 будет составлять точную копию объекта pt, т.к. данные из области памяти pt побайтно копируются в область памяти pt2. В результате указатель coords этих двух разных объектов будет ссылаться на одну и ту же область памяти:



Соответственно, при уничтожении этих объектов, выделенная память освобождается дважды, что приводит к аварийному завершению программы.

Для исправления этой ситуации необходимо при копировании объектов дополнительно выделять память для координат нового объекта.



Но где этот алгоритм копирования прописать в классе PointND? Как раз для этого предназначен еще один конструктор, который называется **конструктор копирования**:

```cpp
PointND(const PointND& other) : total {other.total}
    {
        coords = new int[total];
        set_coords(other.coords, total);
    }
```

Обратите внимание, как он записан. У него один параметр в виде ссылки на объект текущего класса, которую часто записывают, как константную (хотя это не обязательное условие). Именно по этому типу параметра компилятор «понимает», что это конструктор копирования и использует его при создании нового объекта на основе другого того же типа.

В конструкторе копирования мы создаем новый массив координат и заносим в него значения из объекта other. Теперь каждый новый созданный объект класса PointND имеет свой собственный массив координат, который освобождается (в деструкторе) при уничтожении объекта.

Однако обратите внимание, если мы вначале создадим объект, а потом присвоим ему другой:

```cpp
int main()
{
    PointND pt(5);
    PointND pt2;
    pt2 = pt;

    return 0;
}
```

то конструктор копирования здесь срабатывать уже не будет, и мы снова получаем ту же самую проблему, т.к. по умолчанию присваивание одного объекта другому вызывает процедуру побайтного копирования данных. Конструктор копирования вызван не будет, т.к. объект создается с вызовом конструктора по умолчанию. После того, как объект создан, более никакие конструкторы не вызываются.

Забегая вперед, отмечу, что эту операцию также можно переопределить. В самом простом варианте делается это следующим образом:

```cpp
    const PointND& operator=(const PointND& other)
    {
        if(this == &other) return *this; // не присваиваем объект самому себе

        delete[] coords;
```

```
        total = other.total;
        coords = new int[total];
        set_coords(other.coords, total);

        return *this;
    }
```

В результате получаем полноценный класс PointND.

Делегирование конструкторов

Если мы внимательно посмотрим на полученный класс PointND, то увидим в нем некоторое дублирование кода в его конструкторах. Поправить это можно очевидным образом:

```language-cpp
class PointND {
    unsigned total {0};
    int *coords {nullptr};
public:
    PointND() : total(0), coords(nullptr)
        { }
    PointND(unsigned sz) : total(sz)
    {
        coords = new int[total] {0};
    }
    PointND(int* cr, unsigned len) : PointND(len)
    {
//        coords = new int[total];
        set_coords(cr, len);
    }
    PointND(const PointND& other) : PointND(other.coords, other.total)
    {
//        coords = new int[total];
//        set_coords(other.coords, total);
    }
...
};
```

Смотрите, в списке инициализации также допускается вызов конструкторов текущего класса. Такой подход получил название **делегирование конструкторов**, а сами конструкторы, которые в свою очередь вызывают другие, – **делегирующими**. Благодаря делегированию часто удается

сократить дублирование кода при описании классов. Поэтому такой подход не редко используется на практике.